Determinate Imperative Programming

A clocked interpretation of imperative syntax (Extended Abstract)

Vijay Saraswat
IBM Research
vijay@saraswat.org

Radha Jagadeesan
DePaul University
rjagadeesan@depaul.cs.edu

Armando Solar-lezama
UC Berkeley
asolar@eecs.berkeley.edu

Christoph von Praun
IBM Research
praun@us.ibm.com

Abstract

There are a large class of applications, notably those in high-performance computation (HPC), for which parallelism is necessary for performance, not expressiveness. Such applications are typically determinate and have no natural notion of deadlock. Unfortunately, today's dominant HPC programming paradigms (MPI and OpenMP) are based on imperative concurrency and do not guarantee determinacy or deadlock-freedom. This substantially complicates writing and debugging such code.

We present a new concurrent model for mutable variables, the *clocked final* model, CF, that guarantees determinacy and deadlock-freedom. CF views a mutable location as a monotonic stream together with a global *stability rule* which permits reads to stutter (return a previous value) if it can be established that no other activity can write in the current phase. Each activity maintains a local index into the stream and advances it independently as it performs reads and writes. Computation is aborted if two different activities write different values in the same phase.

This design unifies and extends several well-known determinate programming paradigms: single-threaded imperative programs, the "safe asynchrony" of [31], reader-writer communication via immutable variables, Kahn networks, and barrier-based synchronization. Since it is predicated quite narrowly on a re-analysis of mutable variables, it is applicable to existing sequential and concurrent languages, such as Jade, Cilk, Java and X10. We present a formal operational model for a specific CF language, MJ/CF, based on the MJ calculus of [15]. We present an outline of a denotational semantics based on a connection with default concurrent constraint programming. We show that CF leads to a very natural programming style: often an "obvious" shared-variable formulation provides the correct solution under the CF interpretation. We present several examples and discuss implementation issues.

1. Introduction and motivation

For high performance computation, parallelism is a necessary evil – needed for scalability and performance. The algorithms for a large number of problems are determinate and have no natural notion of deadlock. Yet these programs are most commonly expressed in programming models – shared variable concurrency (OpenMP, [1]) and message passing (MPI, [30]) – that make indeterminacy and deadlock all too possible.

This problem is exacerbated in new programming models for high performance languages, such as X10 [7, 26], Titanium [33], Co-Array Fortran (CAF,[22]), UPC [11] and Global Arrays [20] which present an integrated model for shared variables and message passing, based on a *partitioned global address space model*. In this model computations scattered across multiple nodes (e.g. of a single high-performance computer, or a cluster) are viewed logically as running in a common address space, which is lumped into "places" (which host activities with affinity to local data). Aggregate objects (such as arrays) may be distributed across multiple places. Message passing becomes a mechanism for the implementation of remote reads and writes.

Consider the central problem of imperative concurrency. A location L has an initial value v_0. An activity A_1 (the *writer*) wishes to write a value v_1 into this location. Another activity A_2 (the *reader*) wishes to read the value of the location. How is this *read-write* race to be resolved determinately (i.e. in a manner independent of the scheduler)? Similarly, if A_1 and A_2' wish to write to the same location, how is this *write-write* conflict to be resolved?

We consider some paradigmatic examples of concurrent imperative programs which show how reads and writes may be organized to exhibit determinacy and deadlock-freedom

EXAMPLE 1 (CANNON'S ALGORITHM). Consider Canon's algorithm for computing $a \times b$ where a and b are $N \times N$ matrices. First, the ith row of a is left-shifted by i and the jth column of b is up-shifted by j. Then the following computation is performed N times: the value $a[i,j] \times b[i,j]$ is summed into $c[i,j]$ and each a row is left-shifted once and each b column is right shifted one. The following program, written in a Java-like notation is intended to capture this specification. Here `foreach` spawns an activity in parallel for each pair in the given set (`[0:N-1,0:N-1]`), and `finish` causes the initiating activity to wait for their termination. `for` iterates through its elements in sequence.

```
1:  void cannon (double[] c, double[] a, double[] b) {
2:    finish foreach (int i,j in [0:N-1,0:N-1]) {
3:      a[i,j] = a[i, (j+i) % N];
4:      b[i,j] = b[(i+j) % N, j];
5:    }
6:    for (int k in [0:N-1]) {
7:      finish foreach (int i,j:[0:N-1,0:N-1]) {
8:        c[i,j] = c[i,j] + a[i,j] * b[i,j];
9:        a[i,j] = a[i, (j+1) % N];
10:       b[i,j] = b[(i+1) % N, j];
11:     }
12:   }
13: }
```

The program exhibits *parent-child* communication. Each activity spawned on line 2 processes a slice of the two arrays (reading 2 cells and writing 2 cells). The parent waits for these to finish. At line 6 it knows that the shifting has been accomplished, thus data operated on by children activities is now available to the parent. On line 7 it again spawns activities to process other slices of the arrays. On termination of these tasks, the matrix multiplication has been accomplished – the parent activity can access all elements of the array `c`, knowing that none of the activities it had spawned are mutating it.

This specification exhibits read/write conflicts. On Line 3, it is intended that the value of `a[i,(j+i)%N]` is the value *before* the concurrent assignment. A scheduler must not inadvertently schedule the read after the corresponding write. Similarly on lines 8,9, and 10. If these dependencies are respected (e.g. as in Fortran 90, or with the HPF FORALL construct), the program terminates determinately, and without deadlock. As it turns out, this program will operate correctly in the model being proposed in this paper. □

EXAMPLE 2 (*N*-PARTICLE). Given N particles (e.g. molecules) scattered in 3-space and subjet to various mutual forces (e.g. gravitational, electro-static, ionic), the problem is to determine the time evolution of the positions of the particle, given their mass, charge (and other constants) and their initial position. Typically N is large.

While far more sophisticated techniques are used in practice (e.g. Barnes-Hut), we may describe a schematic solution as follows. The points are scattering among processes and computation progresses in a sequence of synchronized phases. In each phase each particle computes the force incident on it at the current time instant, based on the current position of the other particles. This determines the position of particle at the next time instant. When (and only when) each process has computed the position of all particles assigned to it, all processes may move on to the next phase, after redistributing the particles among the processes as necessary (e.g. for load-balancing).

□

This example illustrates the use of mutable arrays operated on in parallel by multiple processes, using *barrier-based* synchronization for determinacy. We think of the array as being indexed by an integer-valued clock. An activity writes a location in phase i of the clock only when all activities have finished writing in phase $i-1$. An activity in phase i reads values produced in phase $i-1$. Thus there are no read-write conflicts. Only one activity is responsible for writing into a specific array location, hence there are no write-write conflicts.

Barrier-based techniques are pervasive in high performance computing. Other examples include relaxation algorithms (e.g. Jacobi) used to solve partial differential equations by applying stencils on array cells in phases. In each phase, an array element is updated based on the values of neighboring array elements specified by the stencil.

EXAMPLE 3 (PIPELINED WAVEFRONTS, UMT2K). The ASCI Purple benchmark, UMT2K, is a deterministic photon transport code for unstructured meshes, based on OpenMP and MPI that exhibits pipelined wavefront computation. Each cell in the mesh receives values from cells "upstream" in the flow (as determined by the mesh geometry), performs computations based on information stored at the cell, and propagates new values to cells "downstream". Multiple cells are allocated to a processor, which remains inactive until the edge of a sweep enters cells on that processor. Typically, multiple sweeps are active at any given time.

Representing the unstructured geometry requires the programming language be able to represent an arbitrary object reference graph. The communication pattern of the wavefront can be easily represented by dataflow. Thus, a language supported (deterministic) dataflow over an arbitrary graph can be used to solve this problem. □

HPC computations are often concerned with parallel reduction operations on (sections of) arrays. These require the scheduling of concurrent associative commutative operations, as illustrated by the following example:

EXAMPLE 4 (HISTOGRAM). The need for *commutative write operations* may be illustrated with the classic histogram problem. Given an integer array `A` containing values from `1` to `n` the problem is to accumulate in parallel in `B[i]` the number of indices in `A` which contain the value `i`. This may be specified quite simply as:

```
1: int[] histogram(int[] A, n) {
2:   final int[] B = new int[1:n];
3:   finish foreach(int i in A) B[A[i]]++;
4:   return B;
5: }
```

Note that according to this code multiple activities may simultaneously perform a commutative update operation on a shared location `B[j]`. (This program will turn out to run correctly in the model being proposed in this paper.) □

Criteria. We are interested in developing a theory of determinate, mutable concurrency. Such a theory must be:

Semantically clean. Computations must be guaranteed to be determinate and deadlock-free

Expressive. The framework must work well within an object-oriented context.

- It must handle the richness of Sequential Java/X10 computations. Thus, it must be possible to create arbitrary object reference graphs, while permitting their manipulation by multiple asynchronous activities.
- It must permit the *same* location to be used repeatedly for communication between multiple reader and writer activities.
- It must handle barrier-based computations, over mutable arrays (cf. N-particle simulations). Read/write operations of array elements should take constant time.
- It must handle synchronous and asynchronous data flow networks (cf. wavefront computations).

General: It must be applicable to arbitrary imperative programming languages.

As Steele points out [31] it may not be possible to design languages that guarantee determinacy and deadlock-freedom statically while still being expressive. Therefore we shall not require that these conditions be established statically. Rather we are interested in

programming models that will detect violations from determinacy dynamically and abort the computation.

1.1 Basic paradigm: the clocked final model

There are some simple answers to this problem that are unfortunately not very general. One answer [31] is that concurrent activities may not access the same location – hence there are no conflicts. The only activities that may coordinate with each other are parent/child activities, at process fork and join. In such a language two operations on a location that are not causally related must be commutative. [31] presents the design of a runtime checker which catches violation of this property and terminates computation abruptly.

This technique is simple and attractive but limited in scope. Another idea is to support *immutable communication*. An activity communicates with another by writing a value into a shared location and declaring that the variable can no longer be mutated. Reads block until a value is written (if ever). There are two major problems with this idea. First, a location can be used only once for communication. A new location must be created for subsequent communication, thus leading to stream-based languages and precluding efficient *reuse* of the same location for repeated communication. Second, such a pattern can easily lead to deadlock, for instance, with a cyclic communication graph.

The* CF *computation model. We propose a simple answer to this problem, which subsumes and generalizes the above two cases.

The essence of imperative programming, the mutable location, provides two guarantees: (1) a read operation returns the value currently in the location, (2) the value in a location does not change unless there is a write. Together, these imply that a read returns the value written by the last write.

This view works well in sequential programming, where the single thread of control ensures that at most one writer or reader is active. With multiple threads of control, read-write and write-write conflicts can arise, as discussed above. Below by a "reader" ("writer") we shall mean an activity that reads (writes). (An activity may both read and write.)

We propose that every mutable location be associated with a *clocked stream* of (immutable, "final") values. The stream is initialized at index 0 with the initial value of the variable. Since a location may have multiple readers and writers, each stream may have multiple readers and writers. (The point to point channels of Kahn networks are the special case with one reader and one writer.) Every activity is equipped with a *world view*, a map from object ids to *object views*, which are maps from fields of the object to read- and write-indices in the stream associated with the field.

An activity's world view is advanced based solely on the operations performed by the activity as follows:

RULE 5 (READ/WRITE). *Initially, the read-index r for a field is set to* 0 *and the write-index w to* 1. *Always $r = w$ or $r = w - 1$. A read returns the value at r; r is incremented and if $r = w$ then so is w. A write writes into $v[w]$; w is incremented and if $r = w - 1$ then so is r.* □

Separate indices for each reader resolves read/read conflicts. Read/write conflicts are resolved by making reads block if the stream is not long enough. A write into a stream adds to the stream at the index of the writer. Write/write conflicts are resolved by the single-assignment rule for each index of the stream: if the write is incompatible with the already existing value (if any), computation aborts globally. Thus as soon as a value is written at an index i, it can be read by another activity at that index (without waiting for other activities to write). This scheme can be generalized to support "commutative writes", Section 2.

EXAMPLE 6 (CANNON, REVISITED). Consider the read/write conflict on Line 3 of Example 1. The activity writing into `a[i,j]` reads the value of `a[i, (j+1)% N]` at index 0 (the initial value), and writes into `a[i,j]` at index 1. On termination the index for `a[i,j]` is `(1,2)` and `a[i, (j+1)% N]` is `(1,1)`. Thus the conflict is resolved in the desired fashion. Similarly for the conflicts on Lines 8 – 10. □

Such a stream approach is general enough to accomodate immutable locations, or even the incremental construction of abritrary higher order value in a purely functional language (cf. work on game semantics [2, 17]). However, it is inadequate to handle mutation: in particular, it does not address the "default assumption" (2) above. This can be seen quite simply: if a stream is of length one, the second read of a reader blocks, so the following program blocks at Line 3:

```
1: x=2;
2: y1=x;
3: y2=x;
```

Thus, the stream approach is not expressive enough: it does not even encompass sequential first-order imperative programming.

Stability. Our key insight is to identify program points where the default (2) can be safely enforced without compromising determinacy. We identify situations (*stabilities*) where the system can perform a *phantom write* extending a stream by one index by copying the value at the last index of the stream. (A read operation that returns a value written by a phantom write is said to *stutter*.) Clearly, to avoid races this can be done only if no writer can write to this index. Note that this immediately gives us sequential imperative programming: in a sequential context, Line 3 is a stable deadlock point in the program above (no activity can write into the stream for `x`). So this rule permits us to copy the value `2` from the first index of stream for `x` into the second, giving us exactly the required behavior.

The solution permits different groups of activities to progress at different, self-timed rates. Assume that an activity A organizes its computation in *episodes*. An episode is sequences of writes followed by a sequence of reads. If positive information is available in the current phase (i.e. some activity has performed a write) these reads return immediately. As far as other activities are concerned, A is busy and *may* produce writes on a set of locations L, hence any attempt to infer that A is *not* going to write to L must block. Only an explicit suspension by A (through an attempt to read values of variables in phases in which the value can not be produced by another writer) is to be viewed as an indication that it has finished writing in this phase. Only when *all* activities that can write on L are so suspended does it become permissible to infer that L will not be written into in this phase. Now this stability can be released by simultaneously performing phantom writes on all the locations in L. This is called the *Stability Rule* and is discussed in more detail in Section 2.1.3.

Thus this approach prefers positive information to negative information (the absence of positive information): as long as an activity is progressing (producing writes for *some* locations) it is considered as active and capable of producing writes for *all* locations. Deadlock is detected and resolved determinately.

Other aspects. We briefly review other aspects of the model. The central novel aspect of the model is the association of locations with streams, and of activities with world views. It becomes necessary to arrange matters so that the world views of activities can be coordinated. The coordination of views at fork and join is handled by:

RULE 7 (FORK/JOIN). *Each activity is initialized with the world view of its parent.*

At join points the world view of the activity is updated with the least upper bound (lub) of the world view of the joining activity.

Intuitively, the lub applies separately to each index, and takes the max. See Section 2 for details, and for a similar rule for lateral communication.

EXAMPLE 8 (CANNON, AGAIN). Recall from Example 6 that on termination of the `(i,j)`th activity launched at Line 2, the index for `a[i,j]` is `(1,2)` and `a[i, (j+i)% N]` is `(1,1)`. Taking lubs, we get the index for the parent activity at Line 6 is `(1,2)` for every location in `a` and `b`. The lub of all of these is `(1,2)`. Similarly on termination of the `(i,j)`th activity spawned on Line 7, the index for `a[i,j]`, `b[i,j]` and `c[i,j]` is `(2,3)` and the other two array elements involved is `(2,2)`. Taking the lubs at Line 11 gives the index for all elements in `a`, `b` and `c` as `(2,3)`. This process is repeated `N` times. With the world view that is returned a read will get the last value written.

Thus the program in Example 1 runs correctly in the CF programming model. □

Thus the clocked final model, CF, smoothly generalizes the basic sequential imperative model while preserving determinacy and deadlock-freedom. A sequential imperative program produces the same result when executed sequentially and when executed as a CF program. CF supports dataflow synchronization, and clock-based synchronization, and permits the same location to be used repeatedly and reliably for writer/reader communications.

This idea is focussed on the structure of the variables and is not tuned to any specific collection of control constructs in the underlying sequential programming paradigm. Thus, existing sequential and parallel languages can be adapted to this model by equipping locations with streams, threads with world views, and reinterpreting fork, join, read and write. Thus, one may speak of C/CF, CAF/CF, HPF/CF, as well as Cilk/CF (Section 2.4.2), X10/CF (Section 2.4.3)etc.

1.2 Contributions of this paper

This paper makes the following contributions:

- We propose a simple, general programming model for determinate imperative concurrency, the *clocked final model* CF, which satisfies the criteria for a theory of determinate imperative concurrency proposed above.
- The model may be used as a basis for designing a deterministic, deadlock-free programming language on top of any sequential imperative language. Existing parallel imperative languages such as Cilk and X10 can be re-interpreted on top of CF "conservatively" (race-free programs have the same results on the same input).
- We present a simple language MJ/CF realizing this model, based on MJ.
- We show several paradigmatic examples can be directly expressed within this framework, and discuss implementation considerations.
- We present a formal operational semantics for MJ/CF, establish determinacy and deadlock-freedom and outline a denotational semantics.

1.3 Comparison with other work

Races are *the* source of non-determinacy in parallel programs [12]. The goal of work on race detection (e.g. [29, 10, 23, 32, 13, 8]) is therefore closely related to the goals of CF. Races are an integral element of the design of data structures such as locks and barriers, hence most research on race detection has focused on *data races* [19]. These occur when data structures that are not specially declared experience unordered conflicting access.

Our work is complementary in nature in that by design the language guarantees a lack of races. Note however that a *MJCF* program may abort because of simultaneous conflicting writes to a location. Techniques from these papers may be used to statically test for these anomalies.

[31] has been an inspiration for our work and presents a design for checking when multiple concurrent activities may perform conflicting write operations on the same location. As discussed above, the design is quite restrictive in that it does not permit reader/writer communication between concurrent siblings.

[4] presents a language design that guarantees no data races. This is accomplished by ensuring that all shared objects are guarded by a monitor. However a program may have determinacy races, i.e. may be indeterminate. Similarly [5] presents a statically checkable design that is race-free but not determinate because of the presence of arbitrary critical sections.

The work of [13, 8] on data race-checking in Cilk is discussed in Section 2.4.2.

Rest of this paper. Next, we discuss the basic model in more detail. We discuss several examples. We discuss implementation considerations. We present a formal operational semantics based on the semantics of X10 and establish determinacy and deadlock freedom.

2. Basic paradigm

2.1 The programming model

2.1.1 Process structure

We shall make the assumption that an activity does not run in parallel with its offspring. This assumption is not essential – and does not curb expressiveness – but makes it easier to state certain conditions below.

When all the activities spawned under a `finish` terminate, the view of the parent activity is updated with the *least upper bound* (lub) of the views of all the offspring activities at their termination. For views v_i, $i < n$, the domain of the lub is the union of the domains of the v_i. The object view o at each oid in the domain maps each field f of o to the max of the read and write indices for $o.f$ for each view v_i defined on o.

These simple rules account for parent/child communication. What about communication between siblings? We adopt the simple idea that a reference to an object o is written into a location by an activity A *together* with the view $v_A(o)$ of o held by A. When this location is read by an activity B, the view $v_B(o)$ is automatically updated to the lub of $v_B(o)$ and $v_A(o)$. This guarantees that B operates on a version of o at least as recent as the one operated on by A.

2.1.2 Streams with a view

A *configuration* should be thought of as a tree of activities and a *store*, recording the *object reference graph* (ORG), i.e. the state of each object. Each object is modeled as a map from the fields of the object to *streams* of values of the right type.

An activity should be thought of as an activation stack, the current expression to be evaluated, together with a *world view* (briefly: view). Given an object o, an *object view* is a map that relates each field of o to a non-negative integer index (called the *version* of the field). A world view is a partial map from object id's to object views.

Activities can simultaneously read and write elements in the stream of each field, according to Rule 5. A read suspends if the value has not yet been written into (or has only been partially writ-

ten into by a commutative write). If the operation is a commutative operation, and all activities which may possibly write into this index have written into it (this may require Stability detection, see below), the value at this index is considered complete and any blocked reads are resumed.

Any attempt to write two different values at an index immediately aborts computation. Since scheduling decisions may affect which of two competing activities last writes into a location, it is not appropriate to consider the last activity to write as the culprit and throw an exception. (This would lead to indeterminacy.)

As stated above, reads may block. This may lead to deadlock, resolved by the *Stability Rule* below.

2.1.3 Progressing from a stable configuration

An activity that tries to read a value that is not yet available will block. In order to adhere to the expectations of the imperative model, however, the reader must be able to read a value even if all the activities which can write to this location have decided not to do so on this phase. When this happens, we say that the location is part of a *stability*; a set of locations that can not be written in the current phase of the clock. The stability is advanced by performing a *phantom write* simultaneously on each location. The phantom write copies into the current write index the value of the corresponding stream at the previous index. We call this the *Stability Rule*, and it causes all reads blocked on this index to stutter and return the value of the previous phase.

The stabilities will be defined formally in terms of a set $N(a)$, an approximation to the set of memory locations reachable from an activity a for writing. We shall have more to say about the specific choice of $N(a)$ in Section 3, but in general the choice should be such that if a is blocked, $N(a)$ can not shirink, and if all the activities that can reach location l are blocked, no new activities can be made to reach l. We will use $N(a)$ to define a predecessor relation among memory locations such that a location l is a *direct predecessor* of location l' if writing to location l can cause a location l' to unblock. This will be the case, for example, if a l' is in $N(a)$ of an activity blocked on l, so writing to l will unblock activity a which could then write to l'.

In order to insure that stabilities are in fact stable, and that the stability rules eliminate all deadlock situations, we will require them to satisfy the following properties.

PROPOSITION 9 (PROPERTIES OF STABILITIES). *Let C be a configuration of the system.*

1. *Let L be a stability in C. Then all activities that can write to L in C are blocked on L.*
2. *Every location l' blocking an activity will have as a predecessor a location that is either in L or that is in the $N(a)$ of some activity a that is not blocked.*

Additionally, stabilities should advance as they appear, which means they should have the the following properties:

PROPOSITION 10 (INDEPENDENCE OF STABILITIES).

1. *Let L be a stability in C and let C' be obtained from C by advancing some other stability L', and letting the unblocked activities evolve. Then L is a stability in C'.*
2. *Any two stabilities in any C are disjoint or identical.*

These properties ensure that stabilities, once formed, cannot grow or shrink, and advancing one stability does not affect others.

Stabilities should also be easy to detect efficiently. In order to achieve this, we will chose a definition of $N(.)$ that allows us to use the following stability rules.

First define the *(private) neighborhood* of an activity a, $P(a)$, to be the set of locations corresponding to the local variables of a and the locations in all those objects o which are reachable only through locations in the neighborhood of a. Let $Pd(a)$ be the union of $P(a')$ for all activities a' that descend from a.

RULE 11. *The following rules may be used to find stabilities:*

Type 1 stability. *When an activity a suspends reading from location l and $l \in P(a)$, $\{l\}$ is a stability.*

Type 2 stability. *If all the descendants of an activity a are blocked on a set of locations $L \subset Pd(a)$, and none of the children of a satisfy this property, L is a stability.*

Type 3 stability. *If all activities in the current configuration are blocked on a set L of locations, and there are no stabilities of type 1 or 2, then L will be a stability.*

2.2 The MJ/CF language

We develop an interpretation of the MJ calculus atop CF to illustrate the power of the CF paradigm. A formal semantics for MJ/CF is provided in Section 3. The core MJ calculus, described in the appendix, includes mutable state, block structured values and basic object-oriented features. It does not however represent packages, import statements, interfaces, arrays, built-in types, method overloading, static state, try/catch/throws, loops, multi-threading. We add to core MJ the following statements. For brevity, our sole commutative write is `int` multiplication.

(Field)
commutative Integer f;

(Statement)
```
finish {async s1,...async sn}
finalize e.f
x *= e;
```

(Promotable exp)
new scoped $C(e_1, \ldots, e_k)$

Scoped variables Say that an object o is *scoped* to an activity A if all references to o are stored either in local variables of A or in objects that are scoped to A. If an object is scoped, then all its locations (i.e. its fields) are scoped. The local variables of an activity are defined to be scoped locations.

MJ/CF permits the creation of scoped objects (`new scoped Point(i,j)`, and has a static type system (via `scoped` annotations on variables) that can track scoped objects and guarantee that they stay scoped. Details are routine and omitted from this extended abstract.

`stuttering`. The function `stuttering()` returns true if the last read performed on a global variable resulted in a stutter.

`finalize L` An activity may assert the statement `finalize L;`. Suppose the activity's index for `L` is `k` and the value of `L` at `k` is `v`. This statement establishes the constraint that the value of `L` in all phases after `k` is going to be `v`. All activities may read `L` in all phases at or after `k` without having to wait for a stability. Any attempt by an activity to assign a value to `L` different from `v` will cause the computation to abort.

The "`final`" variable modifier of Java may be understood in terms of `finalize`.

Repeating reads. Sometimes it is convenient to re-read the last value read from a location `l`. We use the notation `|l|` to refer to the last value read by this activity from the location `l`. That is, this operation does not advance the index in the stream associated with `l`. One may think of `|l|` as a local variable, and any occurrence of `l` in an expression as replaced by the expression `(|l|=l)`. Since `|l|` is a local variable, it may be read multiple times.

2.2.1 Implementation considerations

A stream can be represented as a list or an array. An implementation needs to represent only a *window* on the stream, corresponding to the minimum and maximum indices for the location in the object view of any activity in the configuration. The stream may be run-length encoded, so that a phantom write is implemented by incrementing a stutter-count.

For certain locations it is possible to determine a static bound on the maximal window size. For instance, scoped locations may be implemented as ordinary variables, i.e., variables whose streams have unit windows.

Similarly ancestor-scoped locations that are solely used to communicate between parents and children may be treated as ordinary variables, since they are not subject to concurrent access:

PROPOSITION 12. *A* CF *program that does not exhibit any sibling communication may be executed in buffers with unit windows.*

Since the world views of parents and children are synchronized by `async` and `finish`, the window for each location remain unit-sized.

Single-writer multi-reader variables can be optimized significantly. If all reading activities are known (e.g. through compiler analysis), the runtime system can implement "flow control" techniques. In general, it may not be possible to statically bound the size of the buffers. However, a scheme like the one introduced by [24] can be used to schedule activities in a way that guarantees that the buffers will not grow more than is necessary. A small buffer size is allocated for a shared location, and writers are blocked if necessary. The underlying mechanism for the detection of stabilities can be used to determine if write activities need to be released.

It should be noted that in most scientific computations, communication patterns change very slowly if at all, and therefore the minimum buffer sizes needed to execute the program will be known after the first few iterations.

Program transformation rules. We remark on an important property of the CF model. A location private to an activity can be freely read and written by the activity. This is the basis for many program transformations that can lead to significant performance improvement. In CF, *shared* locations may not be read and written into freely. Indeed, reads and writes are to be thought of as "linear" events (multiplicities matter), that have side-effects on hidden state (the activity's object view). In this CF reflects the reality of modern architectures. The development of a full set of program transformation laws for the CF model is beyond the scope of this paper.

2.3 Programming examples

The programming model offered by MJ/CF is rich enouth to express many usefull patterns including many that are difficult or impossible to express with some of the more restricted programming models discussed earlier.

2.3.1 Stream based computation

Stream based computation in the style of Kahn process networks (KPN) can be difficult to express with models that disalow sibling (lateral) communication. Additionally, KPNs express the end of a computation by deadlocking, which is a challenge for a determinate deadlock free language. For MJ/CF, a deadlock in the KPN translates into a stability in MJ/CF:

EXAMPLE 13 (HAMMING CODE). The problem is to produce a stream of multiples of 2, 3 and 5 in strictly ascending order.

This code prints out the first `N` numbers in the Hamming sequence.

```
class Hamming  {
  int X = 1, X2 = 2, X3 = 3, X5 = 5;
  void run() {
    int one = X; // read and discard first value.
    finish {
      async while (! stuttered(X)) X2 = 2*X;
      async while (! stuttered(X)) X3 = 3*X;
      async while (! stuttered(X)) X5 = 5*X;
      async { // writes X, reads X2, X3, X5.
        int r2 = X2, r3 = X3, r5 = X5;
        for(int i=1;i < N; i++) {
           if (r2==r3) {
              if (r3=<r5) {
                X = r2; r2=X2; r3=X3;
                if (r2==r5) r5=X5;
              } else {
                X=r5;r5=X5;
              }
           } else if (r2 < r3) {
              if (r2 =< r5) {
                X=r2;r2=X2; if (r2 == r5) r5=X5;
              } else {
                X=r5;r5=X5;
              }
           } else {
             if (r3=<r5) {
               X=r3;r3=X3; if (r3==r5) r5=X5;
             } else {
              X=r5;r5=X5
            }}}
      }}}
  int out() {
    return X;
 }
}
```

□

Above, each `async` should be considered as a process in a process network, and each of the variables `X`, `X2`, `X3` and `X5` constitutes a communication channel. Note that after `N` iterations of the `for` loop, there will be a stability involving `X`, and this will cause the first three activities to terminate. Concurrently, another thread may read and use the values produced in `X`.

The program falls in the category of single-writer multiple reader programs discussed earlier. It is possible to schedule the different activities in a data-driven fashion so that only quite small windows are needed for each of the shared variables.

2.3.2 Stencil Computation

Stencil computations are very common in many scientific codes. In general, there is an array where the value at a particular point is defined in terms of the values at neighboring points. The relaxation scheme dictates whether the values to be used should be new values or old values, therefore determining the data dependencies among values.

The code below uses a relaxation scheme that requires the value of `G[i,j]` to be computed in terms of `G[i-1,j]` and `G[i,j-1]` from the current iteration, and `G[i+1,j]`, `G[i,j+1]` `G[i,j]` from the previous iteration.

EXAMPLE 14 (SOR ITERATION). Successive over-relaxation, adapted from the Java Grande Benchmark suite:

```
1: finish foreach( int i,j in [1:M-1, 1:N-1]){
2:   if(i!=1) tmp = G[i-1,j];
3:   if(j!=1) tmp = G[i,j-1];
4:   for (int p=0; p<N; p++)
5:     G[i,j] = omega/4 * (G[i-1,j] + G[i+1,j] + G[i,j-1]
                 + G[i,j+1]) + (1-omega) * G[i,j];
   }
```

□

The loop on Line 1 spawns an activity for each cell in the array G, except for those at the boundary. Lines 2 and 3 guarantee that reads for G[i-1,j] and G[i,j-1] on line 5 will block until the values for the current iteration are available, thereby insuring that the data dependencies are satisfied.

2.3.3 Clocked computation

We sketch an implementation of X10 clocks in MJ/CF. Each clock c is represented as a mutable variable V_c in the scope of the activity creating the clock. A `resume` operation is implemented by performing a commutative write on V_c. A `next` operation is implemented by reading the value of V_c. This suspends until all activities that could write to V_c in the current phase have done so or have suspended. Thus the stability rule will advance activities suspended on a `next` only after all activities have performed the equivalent of a `next`.

2.3.4 Parallel reduction operations

EXAMPLE 15 (HISTOGRAM, REVISITED). The code in Example 4 produces the desired result under CF semantics. All commutative writes to a location B[j] by multiple activities happen in phase 1. On termination of all the `foreach`'s, the current activity's index for each B[j] that was assigned to has moved to 1 (per the Join Rule); hence the location is not available for further commutative writes in phase 1. (Note that no stability was needed to derive this closure.) A subsequent read of B[j] (by the current activity or an activity it spawns) will therefore return the number of indices i in A such that A[i]=j. □

2.3.5 Strongly connected components

In some cases, it is possible to take advantage of stability detection and stuttering to write cleaner algorithms. As an example, consider the following implementation of the algorithm proposed by Fleischer, Hendrickson, Pinar [14] (and improved by McLendon et. al [18]) for finding strongly connected components on distributed graphs.

The algorithm is based on two basic principles. First, given a node v in a graph, consider P the set of all predecessors of v, S the set of all successors, and R be the remainder of the nodes, those that are neither predecessors nor successors. Then, the intersection $P \cap S$ is a strongly connected component, and all other SCCs will be a subset of either S, P or R. Thus, one can find SCCs in a divide and conquer fashion by recursively looking for strongly connected components in $S-P$, $P-S$ and R.

The second basic principle, is that if all the predecessors of v are in a strongly connected component different from v, then we know v must be a singleton.

The main routines in the algorithm are a `trim()` routine that marks nodes as singletons based on the second principle, and a `setSuccessors/setPredecessors` routine which identifies the successors and predecessors for a given node.

The trim routine will mark a node as a singleton when all its predecessors have been marked as belonging to a different scc.

```
0:private void trim(Graph g, int i){
1:  boolean marked[g.size]=false;
2:  finish foreach(n in g.nodes){
3:    if( |n.sccID| == i){
          marked[n.ID];
      } else {
        marked[n.ID]=true;
      }
    }
4:  finish foreach( n in g.nodes){
5:    if( |n.sccID|==i ){
          boolean tmp = true;
6:        for(pred in n.predecessors){
7:            tmp = tmp && marked[pred.ID];
          }
          n.done = tmp;
          if(tmp){
            n.sccID=SINGLETON;
            marked[n.ID]=true;
          }}}}
```

In the code, the array `marked` will indicate whether a node has been marked as belonging to an SCC different from `i`. Initially, all the entries in `marked` corresponding to nodes with `sccID!=i` are set to `true`, but the rest of the entries are made empty, so any thread that tries to read from them will block until they are given a value or until a stability is reached. Note that the read of `n.sccID` in line 2 has been marked with |.| to indicate that it is a non-consuming read.

Once `marked` has been initialized in this way, the foreach in line 3 will mark the singleton nodes according to the second principle defined above. Note that a node with predecessors that have not been marked will block reading on line 6, until all its predecessors have been marked. At some point, there will be a stability involving all the nodes that can not be marked as singleton according to the second principle, and when the stability is resolved, those nodes will read false from line 6, and will not be marked as a singleton.

The code to identify successors of a node is very similar to the code for `trim()`.

```
  private void setSucc(Graph g, Node v, int i){
0:  boolean marked[g.size]=false;
1:  finish foreach(n in g.nodes){
2:    if( |n.sccID| == i ){
        marked[n.ID];
      } else {
        marked[n.ID]=false;
      }
    }
3:  marked[v.ID] = true;
4:  finish foreach(n in g.nodes){
5:    if( marked[n.ID] ){
6:      foreach( t in n.successors){
          if( |t.sccID|==i){
7:          t.isSucc = true;
8:          marked[t.ID] = true;
          }}}}}
```

The key difference is that instead of having to wait until *all* the predecessors have been visited, we want to mark a node as a successor if *any* of its predecessors is a successor. Thus, each node will block on Line 5 until any of its predecessors marks it, and as soon as that happens, it will unblock and mark all its successors in line 8. Once all the successors of v have been marked, all the nodes with `sccID==i` that are not successors of `v` will be involved in a stability on Line 5. When the stability is broken, the finish in line 4 will complete, and the method will return.

2.3.6 Managing Stabilities

Library code must be prepared to handle stabilities whenever it reads a shared variable. We show how to progress from a stability in the following example. It illustrates that intricate patterns of interaction between two siblings are possible in CF.

EXAMPLE 16 (BOUNDED BUFFER). This example illustrates the use of bilateral communication over a single mutable location (`ack`), and how "deadlock" can be detected and handled.

A producer activity P will invoke `push`; a consumer activity C `pop`:

```
OneBuffer o = new OneBuffer();
async produce(o); async consume(o);
```

P must block if it attempts to write a value when the buffer is full' *C* must block if it attempts to read an empty buffer.

The code is:

```
   class OneBuffer {
0:   Object value;
1:   boolean ack;
2:   OneBuffer( Object v) { value = v; ack = true;}
3:   void push(Object v) {
4:      value = v;
5:      boolean c = ack;
6:      while (ack && stuttered());
7:      while (ack && stuttered())  ack = true;
8:   }
9:   Object pop() {
10:     while (ack && stuttered());
11:     ack = true;
12:     while (ack && stuttered()) { ack = true; }
13:     return v;
14:   }
15:}
```

The value to be communicated is stored in `value`; `ack` is used by *P* to signal that a value is available to be consumed, and is also used by *C* to signal that the value has been consumed. Note that `ack` will always take on the value `true`, and is used purely for synchronization.

Consider the code for `push`. While *P* is waiting for the buffer to be read, *C* may be blocked on a read of another variable. This could lead to a stability, and hence the read will return with a stutter (Line 6). These reads must be ignored. Control moves to Line 7 only when a "fresh" value has been received (written by `pop` on Line 11.) Symmetrically, *C* may have been waiting for the buffer to be written into, while *P* was blocked on another read. The resulting stabilities would cause phantom writes that must be skipped over in Line 7 so that a "fresh" write can be performed (to be consumed by `pop` on Line 10). Note that if a new write to an index does not invalidate a phantom write that has already been performed at that index. A subsequent read from that index will continue to cause `stuttered()` to succeed. Thus the idiom on Line 7 (and Line 11) ensures that the current activity can skip over phantom writes and write a "fresh" value (as long as the phantom writes and the new value are the same).

□

2.4 Other CF programming languages

The following examples indicate how CF can augment and usefully extend existing programming languages. CF provides a conceptual framework which can be used to integrate work on data-races, dependency-based control structures and alias control.

2.4.1 Jade/CF

[25] presents a design for *implicitly concurrent* execution of sequential programs that is strikingly similar to our proposal in many respects. First, the design is intended to be applicable to any sequential language. Second, it uses shared variables for communication between activities, rather than some explicit data structure, such as a stream. Third, it distinguishes between private data and data that may be shared between multiple activities. Fourth, its constructs (described below) can be understood purely within an imperative view of computation. Fifth, Jade computations are determinate and deadlock-free.

In Jade, a programmer specifies read-write effects of a task on shared data using the `with`, `withonly` and `without` constructs. `with` specifies positively the side-effects that the enclosed code will perform; this creates synchronization since the implementation must ensure that all prior activities that affect these locations must complete their execution before this code can run. `withonly` specifies that the enclosed code has no side-effects other than the ones specified; before actually mutating data the task must execute a (nested) `with` to obtain access to the data. The code enclosed by a `withonly` can be run as a concurrent activity (`async`). `without` specifies that the current activity will no longer perform the given side-effects; this may permit other dependent tasks to progress. Thus Jade is able to achieve directional, acyclic data flow.

Jade's tasks correspond to CF's `async`s. The `with`, `withonly` and `without` constructs are very synergistic with CF. For instance, `without` information can be used to determine stabilities more locally. `with` provides a novel (determinate, deadlock-free) CF synchronization construct: a preceding task may communicate the "current view" of a location to a succeeding task.

CF's `finish` would be a useful addition to Jade, complementing its data-dependency-based synchronization. Additionally, CF brings to Jade bi-lateral communication, and phased computation. We expect to investigate this integration in future work.

2.4.2 Cilk/CF

Cilk/CF associates every shared memory location with a stream, and every thread with a (world) view. The Cilk thread creation mechanism, `spawn`, creates a thread inheriting the parent view. Parent activity continues to run in parallel with spawned procedures, per Cilk semantics. The `sync` construct updates the parent's world view with the lub of the spawned procedures' world views. Each assignment of a pointer to a global location carries the writer's view of the target location; each read updates the reader's view of the target location.

[13] developed a data race checker for Cilk programs without locks and critical sections. This subset of Cilk programs is shown to be determinate. [8] extended the checker and shows that the introduction of commutative ("Abelian") critical sections do not violate determinacy in the absence of data races. Abelian critical sections are similar to commutative writes of [31].

The basic results about Cilk/CF are as follows.

PROPOSITION 17. *Any* Cilk *program P may be executed as a* Cilk/CF *program. If P is race-free (when viewed as a* Cilk *program), it will produce the same result when executed as a* Cilk/CF *program.*

In particular, race-free Cilk programs will not abort when executed as Cilk/CF programs. The results follow from the observation that race-free Cilk programs do not permit sibling communication, and Proposition 12. Additionally, it is not difficult to see how to design a Cilk/CF implementation so that it can execute a program that is race-free as a Cilk program with buffers of size one.

Note however that Cilk/CF substantially extends the power of Cilk. The presence of lateral communication in Cilk/CF means it is no longer possible to take a Cilk/CF program, strip the Cilk keywords and obtain a C program whose sequential execution is a correct Cilk/CF execution, as the following program demonstrates:

EXAMPLE 18 (Cilk PROGRAMS WITH RACES.). Consider the following program [13, Figure 1]:

```
1: int x;
2: cilk void foo() {
3:   x = x + 1;
4: }
5: cilk int main() {
6:   x = 0;
7:   spawn foo();
8:   spawn foo();
```

```
9:   sync;
10:  printf("x is \%d\n", x);
11:  return 0;
12:}
```

CF forces both the activities spawned at Lines 7–8 to write the value 1 at index 1. After the `sync` the index for the location is 1 and thus the value 1 is printed out, not 2. However, two activities performing a commutative operation `x += 1`, are equivalent to a single activity performing `x += 2`. □

Thus Cilk/CF is more expressive than Cilk because it can determinately express dataflow computation, and lateral and phased communication. This substantially changes the nature of the language.

2.4.3 X10/CF

We are developing an integration of X10 [7, 26] with CF. X10 is a modern statically-typed, class-based object-oriented language providing a notion of places for clustered execution, multi-dimensional arrays, value types and concurrency primitives (`async`, `finish`, clocks, conditional atomic blocks) quite different from Java.

X10/CF augments locations with streams and activities with world views. X10 clocks can be simulated by mutable locations (Section 2.3.3) and are hence not needed in X10/CF as a separate construct.

Another key area of ongoing investigation in X10/CF is the design of a powerful and flexible scoping system for objects which ensures that objects can properly be classified dynamically as private to an activity or shared between them. (Locations known to be scoped can be implemented very efficiently, Section 2.2.1). It must be possible to declare objects as scoped, ensure that they remain scoped during normal operation, and yet permit a way by which the objects may be lent or even published for sharing between multiple activities. It must be possible to "cast" an object to a scoped type; the cast succeeds if in fact the object lies in the current activity's neighborhood.

We expect to build on the static techniques for thread-escape analysis [9] work on alias control [16], such as static techniques using ownership types [4, 5, 6] and alias annotations [3] and dynamic techniques [21].

3. Semantics

We now present a formal semantics for MJ/CF, based on the MJ calculus of [15]. An MJ configuration is a quadruple (H, VS, s, FS). H represents the heap of objects. The heap is represented as a binding of object names to a pair of the class name and a finite function mapping field names to values (objects or basic values). *VS*, the variable stack, represents the block structure of the underlying programming language. The variable stack changes during reduction whenever a new scope is added or removed. s is the statement currently being executed. *FS* the frame stack, represents the continuation that follows the execution of s. In the case that s is an expression that evaluates to a value (say v), the head of the frame stack is an open frame with a hole to indicate the position at which v is to be substituted. Otherwise (s is a statement without a return value), the head of the frame stack is a closed frame without a hole.

This structure is changed for MJ/CF by taking a configuration to be a pair (H, Δ) where H is a heap (changed from MJ to include stream information with each location, ie. fields of every object) and Δ is a tree each of whose nodes is labelled with an activity. An activity is of the form $(s, W, VS, FS, \text{sb})$ where *VS* and *FS* are as above. The component W at a location describes the RWindex of this activity for the location — in terms of the discussion 5, we are coding as a read-index and a "dirty" bit that is 0 or 1; sb is a boolean bit that indicates the stuttering status of the last read. These changes are summarized in Figure 1.

(Configuration)		::=	$(H,\Delta) \mid E$
	Δ	::=	$(W, VS, CF, FS) \mid \Delta \triangleright \bar{\Delta} \mid FS : \Delta$
	$\bar{\Delta}$	::=	$\Delta_1, \ldots, \Delta_n$
(Error)	E	::=	`abort`
(Heap)	H	::=	*finite pf from oids to heap objects*
(Heap Objects)	ho	::=	$(C, \mathcal{F})$
	$\mathcal{F}$	::=	*finite pf from field names to VStreams*
(ValueStream)		::=	*Function from a finite prefix of non−int to ViewedValues x boolean*
(ViewedValues)		::=	*Values x Object View*
(ObjectView)	oW	::=	*finite pf from field names to RWIndx*
(WorldView)	W	::=	*finite pf from oid x field names to RWIndx*
(RWIndx)		::=	*int x [0,1]*

Figure 1. Configurations for MJ/CF

The objects in the heap have some extra structure to support MJ/CF implementation. For every object in the heap, say o, we will assume that there is a boolean slot $o.scoped$. We emphasize that $o.scoped$ is not visible to the programmer. We will assume that the heap contains objects `Integer(i)` for each integer `i`. To simplify presentation, we will assume that these `Integer` objects do not have any fields. `Integer(i).scoped` is always false for all such objects. To simplify notation, we use `Integer(i)` as the object id for `Integer(i)`.

Tree transformations. The transition relation on composite configurations is described as a tree transformation, following our earlier work [26]. Let $\bar{\Delta}$ be the (possibly empty) sequence $\Delta_0, \ldots, \Delta_{n-1}$. We use the syntax $n \triangleright \bar{\Delta}$ to indicate a tree with root node n and subtrees $\Delta_0, \ldots, \Delta_{n-1}$.

A rule $\Delta[\Delta_1] \longrightarrow \Delta[\Delta_2]$ is understood as saying that a tree Δ containing a subtree Δ_1 can transition to a tree which is the same as Δ except that the subtree Δ_1 is replaced by Δ_2. Thus if Δ is the tree $1(2(3,4),5(6))$ then an application of the rule $\Delta[2] \longrightarrow \Delta[8(9)]$ gives the tree $1(8(9,3,4),5(6))$. An application of the rule $\Delta[2 \triangleright \Delta'] \longrightarrow \Delta[8(9)]$ gives the tree $1(8(9),5(6))$ (the entire subtree at 2 is replaced). This rule also ensures that local aborts anywhere in the tree abort the global computation.

$$\frac{(H,\Delta_1) \longrightarrow (H',\Delta_2) \mid E}{(H,\Delta[\Delta_1]) \longrightarrow (H',\Delta[\Delta_2]) \mid E} \text{(COMPOSITE)}$$

MJ ***transitions.*** The transition system incorporates mutatis mutandis all the MJ reduction and decomposition reduction rules ([15], Fig 2,3])) for the various MJ constructs, except for changes caused by the introduction of streamed fields of heap objects. These changes are: the rule (E-New) is replaced by (New) below (to ensure the new object is created with the right initialization for streams); the rules, (E-FieldAccess) and (E-FieldWrite) are replaced by rules that operate appropriately on the streams.

In the development below, we only describe these changes in addition to the description of the new features, namely asynchronous activities and the stuttering rule for stability. The appendix has a full collection of transition rules.

Notation. For any partial function f, we write $f(x) \downarrow$ for $x \in \text{dom}(f)$, $f(x) \uparrow$ for $x \notin \text{dom}(f)$. We use $f[x \mapsto y]$ to describe the partial function that agrees with f everywhere except x where it is set to y. We often use set notation to describe a partial function as a collection of pairs, eg. the partial identity function on positive numbers less than 3 would be written as $\{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2\}$.

If $H(o) = (C, \mathcal{F})$, we write fields(C) (resp. c-fields(C)) or fields(o) (resp. c-fields(o)) for the fieldnames (resp. commutative integer field names) in C. For each commutative integer field f in o, we assume that there is a separate slot (not accessible to user programs) called $curr_f$ that is used to maintain a set of Integers; $*(curr_f)$ returns $(v, \emptyset)$ where v is the the Integer object that is the product of the elements in the set.

For a RWIndex (x,i), we use notation $\mathrm{wInd}((x,i))$ for $x+i$, $\mathrm{rInd}((x,i))$ for x. We write $max((x,i),(y,j))$ for $(max(x,y), max(x+i, y+j) - max(x,y))$, $++_r$ to indicate the partial function that maps (x,i) to $(x+1,0)$, and $++_w$ to indicate the partial function that maps (x,i) to $(x+i,1)$. We use notation $W[o.f{+}{+}_w]$ to indicate the partial function $W[o.f \mapsto {+}{+}_w(W(o.f))]$ (similarly for $W[o.f{+}{+}_r]$)).

Let $dom(\mathcal{F}) = \{f_1, \ldots, f_n\}$. Then: $\texttt{extend}(W,H,o,oW)$ is defined as $\texttt{extendDom}(W,H,o)[o.f_i \mapsto max(oW(f_i), W(o.f_i)) \mid i = 1 \ldots n]$ where $\texttt{extendDom}(W,H,o) = W$ if $H(o) \downarrow$ and $W[o.f_i \mapsto (0,1) \mid i = 1 \ldots n]$ otherwise. We write $\texttt{extend}(W,H,o,0)$ for the special case when oW is the partial function that maps all f_i to $(0,1)$.

Finally, we write $\texttt{comb}(W_1, W_2)$ for the partial function that is undefined if both are undefined, to $max(W_1(o.f), W_2(o.f))$ if both are undefined and and to the sole defined value if only one is defined. $\texttt{comb}(\cdot,\cdot)$ is commutative and associative, so we freely use it with $n > 2$ arguments.

New objects. The new object creation has a standard MJ component described in the first two lines of the hypothesis: fetching the code of the constructor, creating a new object id and creation of a suitable environment to evaluate the body of the constructor. The last two lines of the hypothesis highlight the new features. The first element of the stream associated with the fields is initialized to null (paralleling the initialization of fields to null in MJ). Furthermore, the RWIndex of the streams at the fields of the new object are initialized to $(0,1)$. The commutative integer fields are initialized to $\emptyset$.

(NEW)
$$\frac{\begin{array}{l} cnBody(C) = (\bar{x}, \bar{s}), \Delta_c(C) = \bar{C}, o \notin dom(H), \\ BS = [this \mapsto (o, C), \bar{x} \mapsto (\bar{v}, \bar{C})], \\ o.curr_f = \emptyset, \forall f \in \text{c-fields}(C), o.scoped = \text{ff}, \\ \mathcal{F} = [f[0] \mapsto null, f \in \text{fields}(C)], W' = \texttt{extend}(W,H,o,0) \end{array}}{\begin{array}{l} (H, (W, VS, \texttt{new } C(\bar{v}), FS, \text{sb})) \\ \longrightarrow (H[o \mapsto (C, \mathcal{F})], (W', (BS \circ []) \circ VS, \bar{s}, (return\ o;) \circ FS, \text{sb})) \end{array}}$$

The scoped new constructor is similar to above except for setting $o.scoped = \text{tt}$. The new Integer(i) constructor returns the Integer(i) object from the heap.

Field Write FIELDWRITE1 and FIELDWRITE2 handle writes at already defined indices of the stream at location $o.f$. Such a write succeeds, as per the first rule, if the newly written information agrees with that already existing. On the other hand, by the second rule, an abort is created if there is any mismatch.

(FIELDWRITE1)
$$\frac{\begin{array}{l} H(o) = (C, \mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| > \mathrm{wInd}(W(o,f)), W' = W[o.f{+}{+}_w], \\ \pi_1(\mathcal{F}(f)[\mathrm{wInd}(W(o.f))]) = (v, \{f_i \mapsto W(v.f_i) \mid \forall f_i \in \text{fields}(v)\}); \end{array}}{(H, (W, VS, o.f = v;, FS, \text{sb})) \longrightarrow (H, (W', VS, ;, FS, \text{sb}))}$$

(FIELDWRITE2)
$$\frac{\begin{array}{l} H(o) = (C, \mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| > \mathrm{wInd}(W(o.f)), \\ \pi_1(\mathcal{F}(f)[\mathrm{wInd}(W(o.f))]) \neq (v, \{f_i \mapsto W(v.f_i) \mid \forall f_i \in \text{fields}(v)\}), \end{array}}{(H, (W, VS, o.f = v;, FS, \text{sb})) \longrightarrow \texttt{abort}}$$

The third rule corresponds to the case when the write extends the stream at $o.f$. At every stream location, we have $((v, oW), \text{sb}')$: the actual value v, oW is the writers perspective, ie. the writers RW-indices of the fields of the object v, and sb' in this case is ff since the stream location was created by a program write. See also the read rule below to see the impact of oW, sb'.

(FIELDWRITE3)
$$\frac{\begin{array}{l} H(o) = (C, \mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| = \mathrm{wInd}(W(o.f)), \\ H'(o) = H[o \mapsto (C, \mathcal{F}[f[|\mathcal{F}(f)|] \mapsto ((v, oW), \text{ff})])] \\ oW = \{f_i \mapsto W(v.f_i) \mid \forall f_i \in \text{fields}(v)\}, W' = W[o.f{+}{+}_w]; \end{array}}{(H, (W, VS, o.f = v;, FS, \text{sb})) \longrightarrow (H', (W', VS, ;, FS, \text{sb}))}$$

Finalizing a field is analogous to the third rule, where all *future* indices of the stream are fixed to the current value.

Field Access. Field access can block. The following rule permits field access only if the index of the attempted read is filled in the stream at the location $o.f$. In this case, the RW-index increments by $++_r$.

(FIELDACCESS)
$$\frac{\begin{array}{l} H(o) = (C, \mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| > \mathrm{rInd}(W(o.f)), \\ \mathcal{F}(f)[\mathrm{rInd}(W(o.f))] = ((v, oW), \text{sb}'), \\ W' = \texttt{extend}(W,H,v,oW)[o.f{+}{+}_r] \end{array}}{(H, (W, VS, o.f, FS, \text{sb})) \longrightarrow (H, (W', VS, v, FS, \text{sb}'))}$$

The RW-indices of the fields for the newly read object are updated to account for the writers perspective of the fields of the object being read. The sb of the activity is also updated to reflect the stuttering status of the newly read field.

stuttering(). The stuttering bit is returned.

(STUTTERING)
$$\frac{}{(H, (W, VS, \text{stutter}(), FS, \text{sb})) \longrightarrow (H, (W, VS, \text{sb}, FS, \text{sb}))}$$

Commutative writes. In contrast to regular write (FieldWrite1), commutative writes abort if writes are attempted to an already filled index. In the case when the write is to an as yet unfilled index, the value is added to the set at $curr_f$.

(CFIELDWRITE2)
$$\frac{\begin{array}{l} H(o) = (C, \mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| = \mathrm{wInd}(W(o.f)) \\ v = Integer(y), H'(o) = H[o.curr_f \mapsto \{v\} \cup o.curr_f] \end{array}}{(H, (W, VS, o.f* = v;, FS, \text{sb})) \longrightarrow (H', (W', VS, ;, FS, \text{sb}))}$$

Asynchronous activities. For conceptual simplicity, we break the finish async construct into two pieces. The effect of the combined finish async construct is that a collection of activities are spawned by async, whose termination is awaited by the finish.

async s spawns a new activity initialized with an empty continuation and the variable stack of the spawning environment (the static semantics ensures only final variables can be accessed in *VS*).

(ASYNC)
$$\frac{}{\begin{array}{l} (H, (W, VS, \texttt{async } s, FS, \text{sb})) \\ \longrightarrow (H, (W, VS, ;, FS, \text{sb})) \rhd (W, VS, s, [], \text{sb})) \end{array}}$$

The finish rule creates a nested activity, with the given variable stack but no continuation. On termination of this activity and its subtree the parent activity may continue, with updated W, and restored sb. The calculation of the updated W performs a maximum operation over the RW-indices of the children activities.

(FINISH1)
$$\frac{}{(H, (W, VS, \texttt{finish}(s), FS, \text{sb})) \longrightarrow (H, FS : (W, VS, s, []))}$$

(FINISH2)
Δ is a depth one tree of terminated activities of form $\{(W_1, ;, VS_1, [], \text{sb}_1), (W_2, ;, VS_2, [], \text{sb}_2), \ldots, (W_n, ;, VS_n, [], \text{sb}_n)\}$
$$\frac{}{(H, (FS, \text{sb}) : (VS, ;, []) \rhd \Delta) \longrightarrow (H, (\texttt{comb}(W_1, \ldots, W_n), VS, ;, FS, \text{sb}))}$$

Stability An activity is blocked on location $o.f$ if it is in a configuration of the form $(H, (W, VS, o.f, FS, \text{sb}))$ such that: $H(o) =$

$(C,\mathcal{F}),\mathcal{F}(f)\downarrow,|\mathcal{F}(f)| <= W(o.f)$. An activity is also blocked on location $o.f$ if it is an ancestor of an activity blocked on $o.f$, and is therefore suspended. Given an activity a, denote by $b(a)$ the set of memory locations blocking a.

Recall the definition of $N(a)$ and $P(a)$ from Section 2.1.3. Now, define $Pu(a) = \bigcup_{a' \in \text{ancestor}(a)} P(a')$, the set of locations that are in the neighborhood of a or any of its ancestors. Let pub-l be the locations of public heap objects potentially reachable from multiple threads, hence not in any neighborhood. Then define:

$$N(a) = Pu(a) \cup \text{pub-l}$$

We remark that actions by other activities can't cause $N(a)$ to shrink, and the only way $N(a)$ can grow is if another thread takes an object in its neighborhood and escapes it into the shared heap.

With this notation, we introduce the concept of a predecessor relationship, $\rightarrow$, among memory locations. Let $l \rightarrow l'$ if $l' \in N(a)$ for some activity a blocked on l but not on l' (i.e. $l \in b(a)$ and $l' \notin b(a)$). Let $\stackrel{\star}{\rightarrow}$ be the transitive closure of $\rightarrow$. Informally, $l \rightarrow l'$ in some configuration if adding new information to l (and therefore allowing those threads blocked on l to advance) could cause new information to be added to l'.

The properties of $N(a)$ imply that if $l \rightarrow l'$ it will remain so until some thread writes to l. Furthermore, as long as all the activities that can write to l' are blocked, l' can not acquire new predecessors.

DEFINITION 19 (STABILITY). A set L of locations is said to be *stable* in a configuration when

1. Each $l \in L$ has a thread blocked on it.
2. All threads that can write to an $l \in L$ in the configuration are blocked.
3. $\{l \mid (\exists l' \in L)\ l \rightarrow l'\} \subseteq L$.
4. $(\forall l, l' \in L)\ l \stackrel{\star}{\rightarrow} l',\ l' \stackrel{\star}{\rightarrow} l$

We often just say that L is a *stability* in the configuration. With this definition, propositions 9, 10 and 11 are easily established.

When a stability is detected, two updates are performed. First, the values of the last position in the stream associated with the locations in the stability are copied over to the next index, along with a tt marker to record stuttering. Second, commutative writes are completed: so the temporary values are copied into the stream of the location and the temps are reinitialized for the next round.

(STUTTERING)
$S = \{o_1.f_1, \ldots, o_n.f_n\}$ is a stability in $(H,\Delta), H(o_i) = (C_i, \mathcal{F}_i)$,
$H' = H[o_i \mapsto (C_i,$
$\quad \mathcal{F}[f_i[|\mathcal{F}(f)|] \mapsto (\pi_1(\mathcal{F}[f_i[|\mathcal{F}(f)|-1]]), \text{tt})]) \mid i = 1 \ldots n]$
$H'' = H'[o'.f'_i \mapsto (C_i, \mathcal{F}[f_i[|\mathcal{F}(f)|] \mapsto o'.*(curr_{f'_i}), curr_{f'_i} \mapsto \emptyset)]$

$(H,\Delta) \stackrel{stutter}{\longrightarrow} (H'',\Delta)$

3.1 Determinacy and deadlock-freedom.

The transition relation $(\longrightarrow \cup \stackrel{stutter}{\longrightarrow})$ has no essential critical pairs.

- There are no read-read conflicts since the read-index into the location streams are separate.
- There are no read-write conflicts since reads are blocking, see (FIELDACCESS).
- There are no write-write conflicts since conflicting writes leads to a global abort, see (FIELDWRITE2, COMPOSITE, CFIELDWRITE2).
- There is an inessential critical pair between different uses of (NEW) arising from reuse of the same object id in different activities: this race is resolved by separating the name spaces for object id generation in different activities. In this extended abstract, we elide these routine details.

For stabilities, proposition 10 rules out critical pairs arising from multiple invocations of (STUTTERING). Finally, from condition 2 of the definition of stability 19, there are no conflicts between (STUTTERING) and (FIELDACCESS, *FIELDWRITE*).

In this light, subject to the proviso of separate name-spaces of new object ids for different activities, we get:

PROPOSITION 20. *The transition relation* $(\longrightarrow \cup \stackrel{stutter}{\longrightarrow})$ *is* 1*-step confluent.*

With regard to deadlock detection, consider a globally deadlocked configuration for the transition relation $\longrightarrow$. In the light of (Stuttering), it suffices to argue for the existence of stabilities, to show that progress can be made in the transition relation $(\longrightarrow \cup \stackrel{stutter}{\longrightarrow})$. Since the set of locations is finite, there is a $\rightarrow$-minimal $\stackrel{\star}{\rightarrow}$-equivalence class, which is a stability. So, we get:

PROPOSITION 21. *The transition relation* $(\longrightarrow \cup \stackrel{stutter}{\longrightarrow})$ *is deadlock free .*

3.2 Outline of denotational semantics

We outline a development of the denotational treatment for MJ/CF. The stream interpretation of locations suggests that the denotation should be a function on *perspectives* (a pair consisting of a world view and a heap). A commutative write is modeled by associating that phase of each location with a bag of values, each tagged with a unique id, that depends only on the activation tree of the activity writing that value into the location. With this structure it is not difficult to see that *given* information about the final heap h, to be used to resolve whether or not a read stutters, an activity may be mapped into a closure operator over perspectives in a straightforward fashion, following the structure of the semantics of Default CC [27]. (Note that the world view produced by an activity is not subject to shared mutation.) This observation provides us with the key insight of declarative concurrency: a process should be associated with the set of its fixed points, and parallel composition should be modeled as the intersection of these sets.

An alternate approach is to translate CF programs compositionally into Default CC programs over the Kahn constraint system [28] (augmented with bags for commutative writes). Each activity corresponds to a Default CC process with some private state, namely the world view associated with the process. `finish` is modeled by using the well-known short circuit technique to detect termination.

It remains to show how reads and writes may be modeled. A read returns the value in the stream associated with the given location at the given index – but it also spawns a concurrent agent which uses a default to determine whether the location has been written into in the current phase. If it has not, a phantom write is performed. Note that this agent is spawned only for those locations for which the activity being modeled performs a read. Commutative writers are modeled by adding the value to a bag, and using defaults to "close the bag", and then reducing the bag to obtain the final value.

The above remarks are intended to sketch out the underlying intuitions for the denotational development. We leave for future work the connection between the operational and denotational semantics, and proof principles for establishing properties of CF programs.

4. Conclusion

We have presented a simple re-interpretation of concurrent mutable variables, which leads to a powerful determinate, deadlock-free

model of computation. This model, CF, is applicable to any sequential imperative language, and to many existing (implicitly) concurrent langauges, bringing together synergestically work on race-detection, alias types and dependency-based control structures. We have shown that CF leads to a very natural programming style. We have presented an operational semantics for a language in the CF family, and outlined a denotational interpretation, based on a connection with Default CC.

Acknowledgments

The design of X10 is being done in collaboration with Kemal Ebcioglu, Vivek Sarkar and other members of the X10 team. This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004, and by the National Science Foundation under NSF 0430175.

References

[1] Openmp specifications. www.openmp.org/specs.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for pcf. *Inf. Comput.*, 163(2):409–470, 2000.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 311–330, November 2002.

[4] D. Bacon, R. Strom, and A. Tarafdar. Guava: a dialect of java without data races. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 382–400, New York, NY, USA, 2000. ACM Press.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, November 2002.

[6] C. Boyapati, B. Liskov, and L. Shira. Ownership types for object encapsulation. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'03)*, pages 213–223, June 2003.

[7] P. Charles, C. Grothoff, C. Donawa, K. Ebcioglu, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA*, 2005. To appear.

[8] G.-I Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in cilk programs that use locks. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 298–309, New York, NY, USA, 1998. ACM Press.

[9] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19. ACM Press, November 1999.

[10] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM Press.

[11] T. El-Ghazawi, W. Carlson, and J. Draper. Upc language specification v1.1.1. Technical report, George Washington University, October 2003.

[12] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison, Wisconsin, January 1989.

[13] M. Feng and C. Leiserson. Efficient detection of determinacy races in cilk programs. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 1–11, 1997.

[14] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 505–511, London, UK, 2000. Springer-Verlag.

[15] M.J. Parkinson G.M. Bierman and A.M. Pitts. Mj: An imperative core calculus for java and java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.

[16] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.

[17] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for pcf: I, ii, and iii. *Inf. Comput.*, 163(2):285–408, 2000.

[18] W. Mclendon III, B. Hendrickson, S. Plimpton, and L. Rauchwerger. Finding strongly connected components in parallel in particle transport sweeps. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 328–329, New York, NY, USA, 2001. ACM Press.

[19] R. Netzer and B. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[20] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[21] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Proceedings of the Virtual Machine Research and Technology Symposium (VM'04)*, pages 127–138, May 2004.

[22] R.W. Numrich and J.K. Reid. Co-array Fortran for parallel programming. *Fortran Forum*, 17(2), 1998.

[23] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[24] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berekeley, 1995.

[25] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993.

[26] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Concur '05*, 2005.

[27] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, November–December 1996. Extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[28] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, 1991.

[29] E. Schonberg. On-the-fly detection of access anomalies. *SIGPLAN Not.*, 39(4):313–327, 2004.

[30] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[31] G. L. Steele, Jr. Making asynchronous parallelism safe for the world. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231, New York, NY, USA, 1990. ACM Press.

[32] C. von Praun and T. Gross. Object race detection. In *Conference on Object-Oriented*, pages 70–82, 2001.

[33] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.

A. Syntax

Our presentation is built on top of the MJ calculus. The core MJ calculus, described in the appendix, includes mutable state, block structured values and basic object-oriented features. It does not however represent packages, import statements, interfaces, arrays, built-in types, method overloading, static state, try/catch/throws, loops, multi-threading.

(Program)	p	$::=$	$cd_1 \ldots cd_n; \bar{s}$	
(Class)	cd	$::=$	`class` C `extends` C	
			$\{$ $\text{fd}_1 \ldots \text{fd}_k$ cnd $\text{md}_1 \ldots \text{md}_k$ $\}$	
(Field)	fd	$::=$	$T\ f;$	
(Constructor)	cnd	$::=$	$C(T_1\ x_1, \ldots T_j\ x_j)$	
			$\{$`super`$(e_1, \ldots, e_k); s_1 \ldots s_n$ $\}$	
(Method)	md	$::=$	$\tau\ m(T_1\ x_1, \ldots T_j\ x_j)$	
			$\{s_1 \ldots s_k\}$	
(Return type)	τ	$::=$	T | `void`	
(Type)	T	$::==$	C	
(Expression)	e	$::=$	x	*Variable*
			`null`	*Null*
			$e.f$	*Field access*
			$(C)\ e$	*Cast*
			pe	*Promotable expression*
(Promotable exp)	pe	$::=$	$e.m(e_1, \ldots, e_k)$	*Method invocation*
			`new` $C(e_1, \ldots, e_k)$	*Object creation*
(Statement)	s	$::=$	$;$	*No-op*
			pe	*Promoted expression*
			`if` $(e==e)$ $\{s_1 \ldots s_k\}$	
			`else` $\{s_{k+1} \ldots s_n\}$	*Conditional*
			$e.f = e;$	*Field assignment*
			$T\ x;$	*Local variable declaration*
			$x = e;$	*Variable assignment*
			`return` $e;$	*Return*
			$\{s_1, \ldots, s_n\}$	*Block*

Table 1. Syntax for MJ

The core MJ calculus does not represent packages, import statements, interfaces, arrays, built-in types, method overloading, static state, try/catch/throws, loops, multi-threading. Figure2 describes the features we add to core MJ.

(Field)	
	commutative Integer f;
(Statement)	
	finish {async $s_1, \ldots$ async s_n}
	finalize e.f
	x *= e;
(Promotable exp)	
	`new scoped` $C(e_1, \ldots, e_k)$

Table 2. Additional Syntax for MJ/CF

B. Configurations

An MJ configuration is a quadruple (H, VS, s, FS) where:

- H represents the heap of objects. The heap is represented as a binding of object names to a pair of the class name and a finite function mapping field names to values (objects or basic values).
- VS, the variable stack, represents the block structure of the underlying programming language. The variable stack changes during reduction whenever a new scope is added or removed.
- s is the statement currently being executed.
- FS the frame stack, represents the continuation that follows the execution of s. In the case that s is an expression that evaluates to a value (say v), the head of the frame stack is an open frame with a hole to indicate the position at which v is to be substituted. Otherwise (s is a statement without a return value), the head of the frame stack is a closed frame without a hole.

This structure is changed for MJ/CF by taking a configuration to be a pair (H, Δ) where H is a heap (changed from MJ to include stream information with each location, ie. fields of every object) and Δ is a tree each of whose nodes is labelled with an activity. An activity is of the form (s, W, VS, FS) where VS and FS are as above. The component W describes the read-index of this activity for the streams at all locations. These changes are summarized below.

The objects in the heap have some extra structure to support MJ/CF implementation. For every object in the heap, say o, we will assume that there is a boolean slot $o.scoped$. We emphasize that $o.scoped$ is not visible to the programmer. We will assume that the heap contains objects Integer(i) for each integer i. To simplify presentation, we will assume that these Integer objects do not have any fields. Integer(i).scopedis always false for all such objects. To simplify notation, we use Integer(i) as the object id for Integer(i).

If $H(o) = (C, \mathcal{F})$, we write fields(C) or fields(o) for the fieldnames in C. Similarly, we write c-fields(C) or c-fields(C) for the names of the commuting integer fields. For each commutative Integer field f in o, we assume that there is a separate slot (not accessible to user programs) called $curr_f$ that is used to maintain a set of Integers; $*(curr_f)$ returns $(v, \emptyset)$ where v is the the Integer object that is the product of the elements in the set.

(Configuration)	config	::=	(H,Δ) $\mid E$
	Δ	::=	(W,VS, CF, FS,*sb*) $\mid \Delta \triangleright \bar{\Delta} \mid$ (FS,*sb*):Δ
	$\bar{\Delta}$	::=	$\Delta_1, \ldots, \Delta_n$
(Error)	E	::=	`abort` $\mid$
(Frame Stack)	FS	::=	F ∘ FS $\mid$ []
(Frame)	F	::=	CF $\mid$ OF
(Closed Frame)	CF	::=	$\bar{s} \mid$ `return` e; $\mid \{\} \mid e \mid$ `super`$(\bar{e})$
(Open Frame)	OF	::=	`if`$(\bullet == e)\{\bar{s_1}\}$`else`$\{\bar{s_2}\}$;
			`if`$(v == \bullet)\{\bar{s_1}\}$`else`$\{\bar{s_2}\}$;
			$\bullet.f \mid \bullet.f = e; \mid v.f = \bullet; \mid (C)\bullet$
			$v.m(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)$
			`new`$C(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)$
			`super`$(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)$
			$x = \bullet; \mid$ `return`$\bullet; \mid \bullet.m(\bar{e})$
			finalize $\bullet.f$
(Values)	v	::=	`null` $\mid o$
(Variable Stack)	VS	::=	MS ∘ VS $\mid$ []
(Method Scope)	MS	::=	BS ∘ MS $\mid$ []
(Block Scope)	MS	::=	*finite pf from variables to pairs* (C, v)
(Heap)	H	::=	*finite pf from oids to heap objects*
(Heap Objects)	ho	::=	$(C, \mathcal{F})$
	$\mathcal{F}$	::=	*finite pf from field names to ValueStreams*
(ValueStream)		::=	*Function from a finite prefix of nonnegative integers to ViewedValues x boolean*
(ViewedValues)		::=	*Values x Object View*
(RWIndex)		::=	*int x [0,1]*
(Object View)	oW	::=	*finite pf from field names to RWIndex*
(World View)	W	::=	*finite pf from oid x field names to RWIndex*

Table 3. Configurations for MJ/CF

C. Notation for transition rules

- $eval(MS;x)$, evaluates a variable, x, in a method scope, MS. This partial function is defined only if the variable name is in the scope.
- $update(MS,x \mapsto v)$, that updates a method scope MS with the value v for the variable x, is also a partial function that is undefined if the variable is not in the scope.
- For any partial function f, we write $f(x)\downarrow$ for $x \in \text{dom}(f)$, $f(x)\uparrow$ for $x \notin \text{dom}(f)$.
- We use $f[x \mapsto y]$ to describe the partial function that agrees with f everywhere except x where it is set to y.
- We often use set notation to describe a partial function as a collection of pairs, eg. the partial identity function on positive numbers less than 3 would be written as $\{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2\}$.
- For a RWIndex (x,i), we use notation $\text{wInd}((x,i))$ for $x+i$, $\text{rInd}((x,i))$ for x. We write $\max((x,i),(y,j))$ for $(\max(x,y), \max(x+i, y+j) - \max(x,y))$. We write $\max((x,i),(y,j))$ for $(\max(x,y), \max(x+i,y+j) - \max(x,y))$, $++_r$ to indicate the partial function that maps (x,i) to $(x+1,0)$, and $++_w$ to indicate the partial function that maps (x,i) to $(x+i,1)$.
- Let $\texttt{dom}(\mathcal{F}) = \{f_1, \ldots, f_n\}$. Then:

$$\texttt{extend}(W,H,o,oW) = \texttt{extendDom}(W,H,o)[o.f_i \mapsto \max(oW(f_i), W(o.f_i)) \mid i = 1 \ldots n]$$

  where

$$\texttt{extendDom}(W,H,o) = \begin{cases} W, \text{ if } H(o)\downarrow \\ W[o.f_i \mapsto (0,1) \mid i = 1 \ldots n] \end{cases}$$

  We write $\texttt{extend}(W,H,o,0)$ for the special case when oW is the partial function that maps all f_i to $(0,1)$.

- $$\texttt{comb}(W_1,W_2)(o.f) = \begin{cases} W_1(o.f), \text{ if } W_1(o)\downarrow, W_2(o)\uparrow \\ W_2(o.f), \text{ if } W_2(o)\downarrow, W_1(o)\uparrow \\ (W_1(o.f), W2(o,f)), \text{if } W_1(o)\downarrow, W_2(o)\downarrow \end{cases}$$

  $\texttt{comb}(\cdot,\cdot)$ is commutative and associative, so we freely use it with $n > 2$ arguments.
- For W, a world view, let $W(o.f) = (x,i)$. $W(o.f)++_r$ to indicate the partial function $W[(o.f) \mapsto (x+1,0)]$. We use notation $W(o.f)++_w$ to indicate the partial function $W[(o.f) \mapsto (x+i,1)]$.
- For f a partial function whose domain is a finite prefix, $0 \ldots n$, of the natural numbers, we use notation $\texttt{finalize}(f)$ for:

$$\texttt{finalize}(f)(i) = \begin{cases} f(x), \text{ if } i \leq n \\ f(n), \text{ if } i > n \end{cases}$$

D. Transition system

The transition relation on composite configurations is described as a tree transformation. Let $\bar{\Delta}$ be the (possibly empty) sequence $\Delta_0, \ldots, \Delta_{n-1}$. We use the syntax $n \rhd \bar{\Delta}$ to indicate a tree with root node n and subtrees $\Delta_0, \ldots, \Delta_{n-1}$.

A rule $\Delta[\Delta_1] \longrightarrow \Delta[\Delta_2]$ is understood as saying that a tree Δ containing a subtree Δ_1 can transition to a tree which is the same as Δ except that the subtree Δ_1 is replaced by Δ_2. Thus if Δ is the tree $1(2(3,4),5(6))$ then an application of the rule $\Delta[2] \longrightarrow \Delta[8(9)]$ gives the tree $1(8(9,3,4),5(6))$. An application of the rule $\Delta[2 \rhd \Delta'] \longrightarrow \Delta[8(9)]$ gives the tree $1(8(9),5(6))$ (the entire subtree at 2 is replaced).

Tree rules

(COMPOSITE)
$$\frac{(H,\Delta_1) \longrightarrow (H',\Delta_2) \mid E}{(H,\Delta[\Delta_1]) \longrightarrow (H',\Delta[\Delta_2]) \mid E}$$

(ASYNC)
$$\frac{}{(H,(W,VS,\texttt{async}\ s,FS,\mathrm{sb})) \longrightarrow (H,(W,VS,;,FS,\mathrm{sb})) \rhd (W,VS,s,[],\mathrm{sb}))}$$

(FINISH1)
$$\frac{}{(H,(W,VS,\texttt{finish}(s),FS,\mathrm{sb})) \longrightarrow (H,(FS,\mathrm{sb}) : (W,VS,s,[],\mathrm{sb}))}$$

(FINISH2)
$$\frac{\Delta \text{ is a depth one tree of terminated activities, } \{(W_1,;,VS_1,[],\mathrm{sb}_1),(W_2,;,VS_2,[],\mathrm{sb}_2),\ldots,(W_n,;,VS_n,[],\mathrm{sb}_n)\}}{(H,(FS,\mathrm{sb}) : (VS,;,[]) \rhd \Delta) \longrightarrow (H,(\texttt{comb}(W_1,\ldots,W_n),VS,;,FS,\mathrm{sb}))}$$

D.1 Stability Reduction

Stability

(STUTTERING)
$$\frac{\begin{array}{l} S = \{o_1.f_1,\ldots,o_n.f_n \mid \text{ normal fields}\} \cup \{o'_1.f'_1 \ldots o'_n.f'_n \mid \mathit{commuting fields}\} \text{is a stability in } (H,\Delta), H(o_i) = (C_i,\mathcal{F}_i), \\ H' = H[o_i \mapsto (C_i, \mathcal{F}[f_i[|\mathcal{F}(f)|] \mapsto (\pi_1(\mathcal{F}[f_i[|\mathcal{F}(f)|-1]]),\mathrm{tt}) \mid i = 1\ldots n] \\ H'' = H'[o'.f'_i \mapsto (C_i, \mathcal{F}[f_i[|\mathcal{F}(f)|] \mapsto (o'.*(curr_{f'_i}),\mathrm{ff}), curr_{f'_i} \mapsto \emptyset] \end{array}}{(H,\Delta) \longrightarrow (H'',\Delta)}$$

Reduction1:

(EVAR-ACCESS)
$$\frac{eval(MS,x) = (v,C)}{(H,(W,MS \circ VS,x,FS,\text{sb})) \longrightarrow (H,(W,MS \circ VS,v,FS,\text{sb}))}$$

(EVARWRITE)
$$\frac{eval(MS,x) \downarrow}{(H,(W,MS \circ VS,x = v;,FS,\text{sb})) \longrightarrow (H,(W,(update(MS,x \mapsto v) \circ VS,;,FS,\text{sb}))}$$

(EVARINTRO)
$$\frac{BS' = BS[x \mapsto (null,C)], x \notin dom(BS \circ MS)}{(H,(W,(BS \circ MS) \circ VS,Cx;,FS,\text{sb})) \longrightarrow (H,(W,(BS' \circ MS) \circ VS,;,FS,\text{sb}))}$$

(BLOCKINTRO)
$$\frac{}{(H,(W,MS \circ VS,\{\bar{s}\},FS,\text{sb})) \longrightarrow (H,(W,(\{\} \circ MS) \circ VS,\bar{s},(\{\}) \circ FS,\text{sb}))}$$

(BLOCKELIM)
$$\frac{}{(H,(W,(BS \circ MS) \circ VS,\{\},FS,\text{sb})) \longrightarrow (H,(W,MS \circ VS,;,FS,\text{sb}))}$$

(RETURN)
$$\frac{}{(H,(W,MS \circ VS,return\ v;,FS,\text{sb})) \longrightarrow (H,(W,VS,v,FS,\text{sb}))}$$

(IFELSE1)
$$\frac{v_1 = v_2}{(H,(W,VS,if\ v_1 == v_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};,FS,\text{sb})) \longrightarrow (H,(W,VS,\{\bar{s}_1\},FS,\text{sb}))}$$

(IFELSE2)
$$\frac{v_1 \neq v_2}{(H,(W,VS,if\ v_1 == v_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};,FS,\text{sb})) \longrightarrow (H,(W,VS,\{\bar{s}_2\},FS,\text{sb}))}$$

(FIELDACCESS)
$$\frac{H(o) = (C,\mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| > \text{rInd}(W(o.f)), \mathcal{F}(f)[\text{rInd}(W(o.f))] = ((v,oW),\text{sb}'), W' = \texttt{extend}(W,H,v,oW)[o.f{+}{+}_r]}{(H,(W,VS,o.f,FS,\text{sb})) \longrightarrow (H,(W',VS,v,FS,\text{sb}'))}$$

(FIELDWRITE1)
$$\frac{H(o) = (C,\mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| > \text{wInd}(W(o,f)), \pi_1(\mathcal{F}(f)[\text{wInd}(W(o.f))]) = (v,\{f_i \mapsto W(v.f_i) \mid \text{fields}(v) = \{f_1,\ldots,f_n\}\}), W' = W[o.f{+}{+}_w];}{(H,(W,VS,o.f = v;,FS,\text{sb})) \longrightarrow (H,(W',VS,;,FS,\text{sb}))}$$

(FIELDWRITE2)
$$\frac{H(o) = (C,\mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| > \text{wInd}(W(o.f)), \pi_1(\mathcal{F}(f)[\text{wInd}(W(o.f))]) \neq (v,\{f_i \mapsto W(v.f_i) \mid \text{fields}(v) = \{f_1,\ldots,f_n\}\}),}{(H,(W,VS,o.f = v;,FS,\text{sb})) \longrightarrow \texttt{abort}}$$

(FIELDWRITE3)
$$\frac{H(o) = (C,\mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| = \text{wInd}(W(o.f)), H'(o) = H[o \mapsto (C,\mathcal{F}[f[|\mathcal{F}(f)|] \mapsto ((v,\{f_i \mapsto W(v.f_i) \mid \text{fields}(v) = \{f_1,\ldots,f_n\}\}),\text{ff})], W' = W[o.f{+}{+}_w];}{(H,(W,VS,o.f = v;,FS,\text{sb})) \longrightarrow (H',(W',VS,;,FS,\text{sb}))}$$

(CFIELDWRITE1)
$$\frac{H(o) = (C,\mathcal{F}), \mathcal{F}(f) \downarrow, |\mathcal{F}(f)| > \text{wInd}(W(o.f))}{(H,(W,VS,o.f* = v;,FS,\text{sb})) \longrightarrow \texttt{abort}}$$

(CFIELDWRITE2)
$$\frac{H(o) = (C,\mathcal{F}), \mathcal{F}(f) \downarrow, v = Integer(y), |\mathcal{F}(f)| = \text{wInd}(W(o.f)),, H'(o) = H[o.curr_f \mapsto \{v\} \cup o.curr_f]}{(H,(W,VS,o.f* = v;,FS,\text{sb})) \longrightarrow (H',(W',VS,;,FS,\text{sb}))}$$

(FINALIZE1)
$$\frac{H(o) = (C,\mathcal{F}), (\exists k,k' : \text{wInd}(W(o.f)) \leq k, k' < |\mathcal{F}(f)|)\ \pi_1(\mathcal{F}(f)[k]) \neq \pi_1(\mathcal{F}(f)[k'])}{(H,(W,VS,finalize\ o.f;,FS,\text{sb})) \longrightarrow \texttt{abort}}$$

(FINALIZE2)
$$\frac{H(o) = (C,\mathcal{F}), \mathcal{F}(f) \downarrow, (\forall k,k' : \text{wInd}(W(o.f)) \leq k, k' < |\mathcal{F}(f)|)\ \pi_1(\mathcal{F}(f)[k]) = \pi_1(\mathcal{F}(f)[k']), H'(o) = H[o \mapsto (C,\mathcal{F}[f[|\mathcal{F}(f)|] \mapsto (\texttt{finalize}(\mathcal{F}(f))], W' = W[o.f{+}{+}_w];}{(H,(W,VS,finalize\ o.f;,FS,\text{sb})) \longrightarrow (H',(W',VS,;,FS,\text{sb}))}$$

Reduction2:

(CAST)
$$\frac{H(o) = (C, \mathcal{F}), C \preceq C'}{(H, (W, VS, ((C')o), FS, \text{sb})) \longrightarrow (H, (W, VS, o, FS, \text{sb}))}$$

(NULLCAST)
$$\frac{}{(H, (W, VS, ((C)null), FS, \text{sb})) \longrightarrow (H, (W, VS, null, FS, \text{sb}))}$$

(NEW)
$$\frac{C \neq Integer, cnBody(C) = (\bar{x}, \bar{s}), \Delta_c(C) = \bar{C}, \\ o \notin dom(H), o.scoped = \text{ff}, o.curr_f = \emptyset, \forall f \in \text{c-fields}(C), \mathcal{F} = [f[0] \mapsto null, f \in \text{fields}(C)], \\ BS = [this \mapsto (o, C), \bar{x} \mapsto (\bar{v}, \bar{C})], W' = \texttt{extend}(W, H, o, 0)}{(H, (W, VS, \texttt{new } C(\bar{v}), FS, \text{sb})) \longrightarrow (H[o \mapsto (C, \mathcal{F}), (W', (BS \circ []) \circ VS, \bar{s}, (return\ o;) \circ FS, \text{sb}))}$$

(SCOPED NEW)
$$\frac{C \neq Integer, cnBody(C) = (\bar{x}, \bar{s}), \Delta_c(C) = \bar{C}, \\ o \notin dom(H), o.scoped = \text{tt}, o.curr_f = \emptyset, \forall f \in \text{c-fields}(C), \mathcal{F} = [f[0] \mapsto null, f \in \text{fields}(C)], \\ BS = [this \mapsto (o, C), \bar{x} \mapsto (\bar{v}, \bar{C})], W' = \texttt{extend}(W, H, o, 0)}{(H, (W, VS, \texttt{new } C(\bar{v}), FS, \text{sb})) \longrightarrow (H[o \mapsto (C, \mathcal{F}), (W', (BS \circ []) \circ VS, \bar{s}, (return\ o;) \circ FS, \text{sb}))}$$

(INTEGER NEW)
$$\frac{C = Integer}{(H, (W, VS, \texttt{new } C(v), FS, \text{sb})) \longrightarrow (H, (W', VS, Integer(v), FS, \text{sb}))}$$

(SUPER)
$$\frac{MS(this) = C, C \preceq_1 C', cnBody(C') = (\bar{x}, \bar{s}), \Delta_c(C') = \bar{C}, BS' = [this \mapsto (o, C'), \bar{x} \mapsto (\bar{v}, \bar{C})]}{(H, (W, MS \circ VS, super(\bar{v}), FS, \text{sb})) \longrightarrow (H, (W, (BS' \circ []) \circ (MS \circ VS), \bar{s}, (return\ o;) \circ FS, \text{sb}))}$$

(STUTTERING)
$$\frac{}{(H, (W, VS, \text{stutter}(), FS, \text{sb})) \longrightarrow (H, (W, VS, \text{sb}, FS, \text{sb}))}$$

(METHOD)
$$\frac{H(o) = C, \mathcal{F}), mBody(C, m) = (\bar{x}, \bar{s}), \Delta_m(C)(m) = \bar{C} \rightarrow C', BS' = [this \mapsto (o, C), \bar{x} \mapsto (\bar{v}, \bar{C})]}{(H, (W, VS, o.m(\bar{v}), FS, \text{sb})) \longrightarrow (H, (W, (BS' \circ []) \circ VS, \bar{s}, FS, \text{sb}))}$$

(METHODVOID)
$$\frac{H(o) = C, \mathcal{F}), mBody(C, m) = (\bar{x}, \bar{s}), \Delta_m(C)(m) = \bar{C} \rightarrow void, BS' = [this \mapsto (o, C), \bar{x} \mapsto (\bar{v}, \bar{C})]}{(H, (W, VS, o.m(\bar{v}), FS, \text{sb})) \longrightarrow (H, (W, (BS' \circ []) \circ VS, \bar{s}, (return\ o;) \circ FS, \text{sb}))}$$

(SKIP)
$$\frac{}{(H, (W, VS, ;, F \circ FS, \text{sb})) \longrightarrow (H, (W, VS, F, FS, \text{sb}))}$$

(SUB)
$$\frac{}{(H, (W, VS, v, F \circ FS, \text{sb})) \longrightarrow (H, (W, VS, ;, F[v] \circ FS, \text{sb}))}$$

Decomposition Reduction1:

(SEQ)

$$(H,(W,VS,s_1 s_2 \ldots s_n,FS,\text{sb})) \longrightarrow (H,(W,VS,s_1,(s_2 \ldots s_n)\circ FS,\text{sb}))$$

(RET)

$$(H,(W,MS\circ VS,return\ e;,FS,\text{sb})) \longrightarrow (H,(W,MS\circ VS,e,(return\ \bullet;)\circ FS,\text{sb}))$$

(EXPSTATE)

$$(H,(W,MS\circ VS,e';,FS,\text{sb})) \longrightarrow (H,(W,MS\circ VS,e',FS,\text{sb}))$$

(IF1)

$$(H,(W,VS,if\ e_1 == e_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};,FS,\text{sb})) \longrightarrow (H,(W,VS,e_1,(if\ \bullet == e_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};)\circ FS,\text{sb}))$$

(IF2)

$$(H,(W,VS,if\ v_1 == e_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};,FS,\text{sb})) \longrightarrow (H,(W,VS,e_2,(if\ v_1 == \bullet\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};)\circ FS,\text{sb}))$$

(FIELDACCESS)

$$(H,(W,VS,e.f;,FS,\text{sb})) \longrightarrow (H,(W,VS,e,(\bullet.f)\circ FS,\text{sb}))$$

(FIELDWRITE1)

$$(H,(W,VS,e.f = e';,FS,\text{sb})) \longrightarrow (H,(W,VS,e,(\bullet.f = e';)\circ FS,\text{sb}))$$

(FIELDWRITE2)

$$(H,(W,VS,v.f = e;,FS,\text{sb})) \longrightarrow (H,(W,VS,e,(v.f = \bullet;)\circ FS,\text{sb}))$$

(FINALIZE)

$$(H,(W,VS,finalize\ e.f;,FS,\text{sb})) \longrightarrow (H,(W,VS,e,(finalize\ \bullet.f;)\circ FS,\text{sb}))$$

(VARWRITE)

$$(H,(W,VS,x = e;,FS,\text{sb})) \longrightarrow (H,(W,VS,e,(x = \bullet;)\circ FS,\text{sb}))$$

(CAST)

$$(H,(W,VS,(C)e,FS,\text{sb})) \longrightarrow (H,(W,VS,e,((C)\bullet)\circ FS,\text{sb}))$$

Decomposition Reduction2:

(NEW-I)

$$(H,(W,VS,new\ C(v_1,\ldots,v_{i-1},e_i,\ldots,e_n),FS,\text{sb})) \longrightarrow (H,(W,VS,e_i,(new\ C(v_1,\ldots,v_{i-1},\bullet,\ldots,e_n)\circ FS,\text{sb}))$$

(SUPER-I)

$$(H,(W,VS,super.(v_1,\ldots,v_{i-1},e_i,\ldots,e_n),FS,\text{sb})) \longrightarrow (H,(W,VS,e_i,(super.(v_1,\ldots,v_{i-1},\bullet,\ldots,e_n)\circ FS,\text{sb}))$$

(METHOD1)

$$(H,(W,VS,e.m(e_1,\ldots,e_n),FS,\text{sb})) \longrightarrow (H,(W,VS,e,(\bullet.m(e_1,\ldots,e_n)\circ FS,\text{sb}))$$

(METHOD2)

$$(H,(W,VS,v.m(v_1,\ldots,v_{i-1},e_i,\ldots,e_n),FS,\text{sb})) \longrightarrow (H,(W,VS,e_i,(v.m(v_1,\ldots,v_{i-1},\bullet,\ldots,e_n)\circ FS,\text{sb}))$$